REMARKS/ARGUMENTS

Claims 1-24 are rejected under 35 U.S.C. 102(e) as being anticipated by U.S. Patent No. 5,742,782 issued to Ito et al. ("Ito"). Claims 1-24 remain in this application. Claims 1, 8, 13, and 20 have been amended. Applicants respectfully submit that claims 1-24 overcome the anticipation rejection based on Ito.

Claim 1

As amended, claim 1 recites:

"a single instruction that specifies an operation <u>to cause multiple instances of the operation to be performed, each instance of the operation to be performed using a different one of the plurality of data elements</u> in partitioned fields of at least one of the registers to produce a catenated result" (emphasis added)

The amended claim language of claim 1 requires that a single instruction specify an operation to cause multiple instances of the operation to be performed, each instance to be performed using a different one of the plurality of data elements in partitioned fields of at least one of the registers. For example, in one embodiment of the invention, a "Group Add" instruction is carried out with a 128-bit register that contains sixteen individual 8-bit operands in partitioned fields. That is, the "Group Add" instruction specifies an ADD operation, and sixteen instances of the ADD operation are performed, each instance being performed on one of the sixteen individual 8-bit operands. Accordingly, sixteen different individual add operations may take place as the result of a single "Group Add" instruction. See specification at ¶ 0221.

By contrast, Ito discloses a very long instruction word (VLIW) instruction that specifies four operations, each to be performed in a traditional manner. That is, only one instance of each operation is to be performed. Fig. 8 of Ito illustrates the VLIW format:

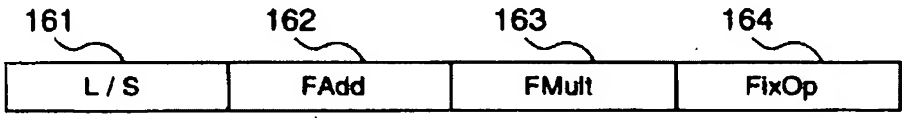


Fig. 8 of Ito

As seen in this figure, the VLIW format contains four different operations: (1) a load/store (L/S) operation, (2) a floating-point add (FAdd) operation, (3) a floating-point multiply (FMult) operation, and (4) a fixed-point operation (FixOp). Each of these four operations is performed only once. In a sense, Ito merely teaches that four conventional instructions can be specified together, using the "very long" format. Nowhere in the disclosure of Ito is there any teaching or even remote suggestion that an operation can be specified to cause multiple instances of the operation to be performed.

In fact, Ito actually teaches away from the invention as recited in claim 1. To perform one instance of an operation, Ito requires that the operation be specified in a VLIW instruction. To perform another instance of the same operation, Ito requires the operation be specified again in a VLIW instruction. Thus, to perform sixteen different instances of an ADD operation, for example, Ito would require the ADD operation to be specified sixteen separate times. This is the antithesis of a single instruction that specifies an operation to cause multiple instances of the operation to be performed, each instance of the operation to be performed using a different one of the plurality of data elements, as recited in claim 1. Accordingly, Applicants respectfully submit that Ito cannot anticipate claim 1.

<u>Claim 6</u>

Claim 6 depends from claim 1 and therefore incorporates all of its limitations. For at least the reasons discussed above with respect to claim 1, Ito fails to anticipate claim 6.

Ito also fails to anticipate claim 6 for the additional reason that Ito does not disclose a virtual memory addressing unit. Claim 6 recites:

> **"The processor of claim 1 further comprising a <u>virtual memory addressing unit</u> and a cache operable to store data communicated between the external interface and the data path" (emphasis added)**

A claim is anticipated only if each and every element as set for the in the claim is found, either expressly or inherently described, in a single prior art reference. MPEP § 1231. Here, the Office Action presents an anticipation rejection of claim 6 based on Ito. However, Ito contains absolutely no mention of a virtual memory addressing unit, which is clearly recited in claim 6. That is, Ito fails to explicitly disclose a virtual memory addressing unit.

The issue then is whether Ito "inherently" discloses a virtual memory addressing unit. Inherency can only be established if a feature is necessarily present, even though it is not explicitly disclosed by a reference. Inherency may not be established by probabilities or possibilities. The mere fact that a certain thing may result from a given set of circumstances is not sufficient. As stated in MPEP § 2112(IV):

> **The fact that a certain result or characteristic may occur or be present in the prior art is not sufficient to establish the inherency of that result or characteristic. In re Rijckaert, 9 F.3d 1531, 1534, 28 USPQ2d 1955, 1957 (Fed. Cir. 1993)..."To establish inherency, the extrinsic evidence 'must make clear that the missing descriptive matter is necessarily present in the thing described in the reference, and that it would be so recognized by persons of ordinary skill. Inherency, however, may not be established by probabilities or possibilities. The mere fact that a certain thing may result from a given set of circumstances is not sufficient.' " In re Robertson, 169 F.3d 743, 745, 49 USPQ2d 1949, 1950-51 (Fed. Cir. 1999).**

The Office Action presents arguments that fail to meet the standard for establishing inherency. The Office Action alleges that Ito's disclosures "can" be considered to be virtual memory addressing. Specifically, the Office Action states: "Any communication with an external device (with an external memory space) can be considered to be virtual memory addressing. Further, accessing main memory can be considered using virtual memory (with respect to the internal registers)." Present Office Action dated 2/9/07, p. 7, ¶ 16 (emphasis added). However, merely alleging the probability or possibility that Ito's features "can" be considered virtual memory addressing fails to establish that Ito's disclosures necessarily discloses virtual memory addressing.

Indeed, there exist many well-known computer architectures that do NOT implement virtual memory. For example, early Apple Macintosh systems, built on the Mac OS operating system and the 68000 processor, controlled the assignment and usage of the system's 128K of physical memory (RAM). See Mac OS memory management, <http://en.wikipedia.org/wiki/Mac_OS_memory_management>, ¶¶ 1-2 (last visited May 15,

207) (Attachment A). No virtual memory was implemented. Instead, the Macintosh system simply optimized the use of its limited 128K of physical memory. See id., ¶ 2. As another example, current PC-based computer systems based on the Microsoft Windows 95 operating system in some instances allow virtual memory to be "turned off." See Microsoft Help and Support, <http://support.microsoft.com/kb/q128327/>, "More Information" section, ¶ 3 (last visited May 15, 2007) (Attachment B). Clearly in these examples, normal accesses to external devices and main memory continue to occur, even though no virtual memory is used. Thus, access of external devices and main memory does not necessarily imply use of virtual memory. Similarly, Ito's alleged access of external devices and main memory does not necessarily indicate any use of virtual memory. As such, Ito fails to inherently disclose a virtual memory addressing unit, as recited in claim 6. For all of the reasons stated above, Ito fails to anticipate claim 6.

Claims 2-5 and 7

Claims 2-5 and 7 depend from claim 1 and therefore incorporates all of its limitations. As such, for at least the reasons discussed above with respect to claim 1, Ito also fails to anticipate claims 2-5 and 7.

Claim 8

Claim 8 is rejected based on the same rationale as claim 1. For at least similar reasons as stated above with respect to claim 1, Ito also fails to anticipate claim 8.

Claims 9-12

Claims 9-12 depend from claim 8 and therefore incorporate all of its limitations. As such, for at least the reasons discussed above with respect to claim 8, Ito also fails to anticipate claims 9-12.

Claim 13

Claim 13 is rejected based on the same rationale as claim 1. For at least similar reasons as stated above with respect to claim 1, Ito also fails to anticipate claim 13.

Claims 14-17

Claims 14-17 depend from claim 13 and therefore incorporate all of its limitations. As such, for at least the reasons discussed above with respect to claim 13, Ito also fails to anticipate claims 14-17.

Claim 18

Claim 18 depends from claim 13 and therefore incorporates all of its limitations. As such, for at least the reasons discussed above with respect to claim 13, Ito also fails to anticipate claim 18. Furthermore, claim 18 is also rejected based on the same rationale as claim 6. Thus, Ito fails to anticipate claim 18 for at least similar reasons as stated above with respect to claim 6, as well.

Claim 19

Claim 19 depends from claim 13 and therefore incorporates all of its limitations. As such, for at least the reasons discussed above with respect to claim 13, Ito also fails to anticipate claim 19.

Claim 20

Claim 20 is rejected based on the same rationale as claim 1. For at least similar reasons as stated above with respect to claim 1, Ito also fails to anticipate claim 20.

Claims 21-24

Claims 21-24 depend from claim 20 and therefore incorporate all of its limitations. As such, for at least the reasons discussed above with respect to claim 20, Ito also fails to anticipate claims 21-24.

CONCLUSION

In view of the above amendments and remarks, Applicants submit that this application should be allowed and the case passed to issue. If there are any questions regarding this

Amendment or the application in general, a telephone call to the undersigned would be appreciated to expedite the prosecution of the application.

To the extent necessary, a petition for an extension of time under 37 C.F.R. 1.136 is hereby made. Please charge any shortage in fees due in connection with the filing of this paper, including extension of time fees, to Deposit Account 500417 and please credit any excess fees to such deposit account.

Respectfully submitted,

McDERMOTT WILL & EMERY LLP

[illegible]

Demetria A. Buncum
Registration No. 58,848

600 13th Street, NW
Washington, DC 20005-3090
Phone: 202.756.8000 DAB
Facsimile: 202.756.8087
Date: June 8, 2007

Please recognize our Customer No. 20277 as our correspondence address.

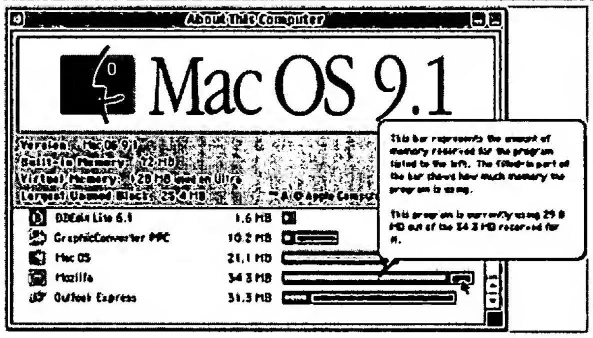
Attachment A - Mac OS memory management

Mac OS memory management

From Wikipedia, the free encyclopedia

Historically, the Mac OS used a form of memory management that has fallen out of favour in modern systems. Criticism of this approach was one of the key areas addressed by the change to Mac OS X.

The original problem for the designers of the Macintosh was how to make optimum use of the 128 kB of RAM that the machine was equipped with. Since at that time the machine could only run one application program at a time, and there was no permanent secondary storage, the designers implemented a simple scheme which worked well with those particular constraints. However, that design choice did not scale well with the development of the machine, creating various difficulties for both programmers and users.



Version: Mac OS 9.1
Built-in Memory: 72 MB
Virtual Memory: 128 MB used on Ultra
Largest Unused Block: 25.4 MB

DDCalc Lite 6.1	1.6 MB
GraphicConverter PPC	10.2 MB
Mac OS	21.1 MB
Mozilla	34.3 MB
Outlook Express	31.3 MB

This bar represents the amount of memory reserved for the program listed to the left. The filled-in part of the bar shows how much memory the program is using.

This program is currently using 29.8 MB out of the 34.3 MB reserved for it.

"About This Computer" Mac OS 9.1 window showing the memory consumption of each open application and the system software itself.

Contents

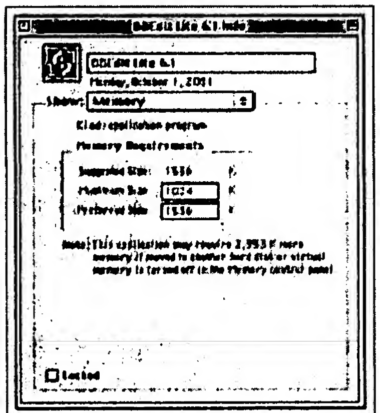
Fragmentation

The chief worry of the original designers appears to have been fragmentation—that is, repeated allocation and deallocation of memory through pointers leads to many small isolated areas of memory which cannot be used because they are too small, even though the total free memory may be sufficient to satisfy a particular request for memory. To solve this, Apple designers used the concept of a relocatable handle, a reference to memory which allowed the actual data referred to be moved without invalidating the handle. Apple's scheme was simple - a handle was simply a pointer into a (non relocatable) table of further pointers, which in turn pointed to the data. If a memory request required compaction of memory, this was done and the table, called the master pointer block, was updated. The machine itself implemented two areas in the machine available for this scheme - the system heap (used for the OS), and the application heap. As long as only one application at a time was run, the system worked well. Since the entire application heap was dissolved when the application quit, fragmentation was minimized.

However, in addressing the fragmentation problem, all other issues were overlooked. The system heap was not protected from errant applications, and this was frequently the cause of major system problems and crashes. In addition, the handle-based approach also opened up a source of programming errors, where pointers to data within such relocatable blocks could not be guaranteed to remain valid across calls that might cause memory to move. In reality this was almost every system API that existed. Thus the onus was on the programmer not to create such pointers, or at least manage them very carefully. Since many programmers were not generally familiar with this approach, early Mac programs suffered frequently from faults arising from this - faults that very often went undetected until long after shipment.

Palm OS uses a similar scheme for memory management. However, the Palm version makes programmer error more difficult. For instance, in Mac OS to convert a handle to a pointer, a program just de-references the handle directly. However, if the handle is not locked, the pointer can become invalid quickly. Calls to lock and unlock handles are not balanced; ten calls to HLock are undone by a single call to HUnlock. In Palm OS, handles are opaque type and must be de-referenced with MemHandleLock. When a Palm application is finished with a handle, it calls MemHandleUnlock. The Palm OS keeps a lock count for blocks; after three calls to MemHandleLock, a block will only become unlocked after three calls to MemHandleUnlock.

MultiFinder


[illegible] Info

[illegible]

Monday, October 1, 2001

Show: Memory

Kind: application program

Memory Requirements

Suggested Size: 1536 K

Minimum Size: 1024 K

Preferred Size: 1536 K

Note: This application may require 2,353 K more memory if moved to another hard disk or virtual memory is turned off in the Memory control panel

Locked

On classic Mac OS (from version 7 to 9), each application had a configurable but fixed amount of memory reserved for it.

The situation worsened with the advent of MultiFinder, which was a way for the Mac to run multiple applications at once. This was a necessary step forward for users, who found the one-app-at-a-time approach very limiting. However, because Apple was now committed to its memory management model, as well as compatibility with existing applications, it was forced to adopt a scheme where each application was allocated its own heap from the available RAM. The amount of actual RAM allocated to each heap was set by a value coded into each application, set by the programmer. Invariably this value wasn't enough for particular kinds of work, so the value setting had to be exposed to the user to allow them to tweak the heap size to suit their own requirements. This exposure of a technical implementation detail was very much against the grain of the Mac user philosophy. Apart from exposing users to esoteric technicalities, it was inefficient, since an application would grab (unwillingly) all of its allotted RAM, even if it left most of it subsequently unused. Another application might be memory starved, but was unable to utilise the free memory "owned" by another application.

MultiFinder became the Finder in System 7, and by then the scheme was utterly entrenched. Apple made some attempts to work around the obvious limitations - temporary memory was one, where an application could "borrow" free RAM that lay outside of its heap for short periods, but this was unpopular with programmers so it largely failed to solve the problem. There was also an early virtual memory scheme, which attempted to solve the issue by making more memory available by paging unused portions to disk, but for most users with 68K Macintoshes, this did nothing but slow everything down without solving the memory problems themselves. By this time all machines had permanent hard disks and MMU chips, and so were equipped to adopt a far better approach. For some reason, Apple never made this a priority until Mac OS X, even though several schemes were suggested by outside developers that would retain compatibility while solving the overall memory management problem. Third party replacements for the Mac OS memory manager, such as Optimem, showed it could be done.

32-bit clean

Originally the Macintosh had 128 kB of RAM, with a limit of 512 kB. This was increased to 4 MB upon the introduction of the Macintosh Plus. These Macintosh computers used the 68000 CPU, a 32-bit processor, but only had 24 physical address lines. The 24 lines allowed the processor to address up to 16 MB of memory (2^{24} bytes), which was seen as a sufficient amount at the time. However, the RAM limit in the Macintosh design was 4 MB of RAM and 4 MB of ROM, because of the structure of the memory map [1] (http://www.osdata.com/system/physical/memmap.htm#MacPlus).

Because memory was a scarce resource, the authors of the Mac OS decided to take advantage of the unused byte in each address. The original Memory Manager (up until the advent of System 7) placed flags in the high 8 bits of each 32-bit pointer and handle. Each address contained flags such as "locked", "purgeable", or "resource", which were stored in the master pointer table. When used as an actual address, these flags were masked off and ignored by the CPU.

While a good use of very limited RAM space, this design led to problems once Apple introduced the Macintosh II, which used the 32-bit Motorola 68020 CPU. The 68020 had 32 physical address lines and could address up to 4GB (2^{32} bytes) of memory. The flags that the Memory Manager stored in the high byte of each pointer and handle were significant now, and could lead to addressing errors.

In theory, the architects of the Macintosh system software were free to change the "flags in the high byte" scheme to avoid this problem, and they did. For example, on the Macintosh II, HLock() was rewritten to implement handle locking in a way other than flagging the high bits of handles. However, many Macintosh application programmers — and a great deal of the Macintosh system software code itself — accessed the flags directly rather than use the APIs, such as HLock(), which had been provided to manipulate them. By doing this they rendered their applications incompatible with true 32-bit addressing, and this became known as not being "32-bit clean".

Four Macintosh models were manufactured with "dirty ROMs" that contained non 32-bit clean software: the Macintosh II, IIx, IIcx, and SE/30. In order to stop continual system crashes caused by this issue, the Mac OS running on any Macintosh with dirty ROMs would force the CPU into 24-bit mode, and would only recognize and address the first 8 megabytes of RAM, an obvious flaw in machines whose hardware was wired to accept up to 128MB RAM — and whose product literature advertised this capability. Surprisingly, the first solution to this flaw was published by software utility company Connectix, whose 1991 product MODE32 replaced every one of Apple's "dirty" system software routines, making the system software 32-bit clean and enabling the use of all the RAM in the machine. Apple licensed the software from Connectix later in 1991 and distributed it for free. The Macintosh IIci and later computers had 32-bit clean ROMs.

However it was quite a while before applications were updated to remove all 24-bit dependencies, and System 7 provided a way to switch back to 24-bit mode if application incompatibilities were found. By the time of migration to the PowerPC and System 7.1.2, 32-bit cleanliness was mandatory for creating native applications.

Object orientation

The rise of object-oriented languages for programming the Mac — first Object Pascal, then later C++ — also caused problems for the memory model adopted. At first, it would seem natural that objects would be implemented via handles, to gain the advantage of being relocatable. However, these languages, as they were originally designed, used pointers for objects, which would lead to fragmentation issues. A solution, implemented by the THINK (later Symantec) compilers, was to use Handles internally for objects, but use a pointer syntax to access them. This seemed a good idea at first, but soon deep problems emerged, since programmers could not tell whether they were dealing with a relocatable or fixed block, and so had no way to know whether to take on the task of locking objects or not. Needless to say this led to huge numbers of bugs and problems with these early object implementations. Later compilers did not attempt to do this, but used real pointers, often implementing their own memory allocation schemes to workaround the Mac OS memory model.

While the Mac OS memory model, with all its inherent problems, remained this way right through to Mac OS 9, due to severe application compatibility constraints, the increasing availability of cheap RAM meant that by and large most users could upgrade their way out of a corner. The memory wasn't used efficiently, but it was abundant enough that the issue never became critical. This is perhaps ironic given that the purpose of the original design was to maximise the use of very limited amounts of memory. Mac OS X finally does away with the whole scheme, implementing a modern sparse virtual memory scheme. A subset of the older memory model APIs still exist for compatibility as part of Carbon, but map to the modern memory manager (a threadsafe malloc implementation) underneath. Apple recommends that Mac OS X code use malloc and free "almost exclusively" (see Memory Allocation Recommendations on Mac OS X (http://developer.apple.com/technotes/tn2005/tn2130.html)).

Retrieved from "http://en.wikipedia.org/wiki/Mac_OS_memory_management"

Categories: Mac OS | Memory management

Attachment B - http support Microsoft

How Windows 95 Manages Virtual Memory

This article was previously published under Q128327

Article ID	: 128327
Last Review	: November 15, 2006
Revision	: 2.1

If this article does not describe your hardware-related issue, please see the following Microsoft Web site to view more articles about hardware:

http://support.microsoft.com/default.aspx/w98?sid=460
(http://support.microsoft.com/?scid=http%3a%2f%2fsupport.microsoft.com%2fdefault.aspx%2fw98%3fsid%3d460)

On This Page

SUMMARY

This article contains information and commonly asked questions about virtual memory in Microsoft Windows.

Windows uses a dynamic virtual memory manager to handle swap file duties. You should use the default virtual memory settings whenever possible. However, if you have limited hard disk space you may want to set some of the virtual memory settings manually.

MORE INFORMATION

In order to provide more memory to applications than is physically present in the computer in the form of RAM, Windows uses hard disk space to simulate RAM. The amount of RAM in the computer plus the size of the paging file (also known as the swap file) equals the total physical memory, or virtual memory, size. Windows uses a dynamic paging file that remains at a size of 0K until it is needed. The paging file can grow to use all the available space on the hard disk if it is necessary. This is the default setting for the paging file. You should use this setting if possible.

If you have limited hard disk space, other applications may reduce the amount of virtual memory below that needed by Windows and its applications. If this occurs, choose the "Let me specify my own virtual memory settings" option button on the Performance tab of the My Computer property sheet. You can use this option to set a minimum and maximum size for the paging file. Setting this too low can result in "out of memory" errors or worse when virtual memory requirements exceed the maximum limit.

You can also choose to use no virtual memory. You should use this option only if there is enough RAM to meet all the needs of Windows and the applications you run.

NOTE: The Disable Virtual Memory option is unavailable on computers with 8 MB of RAM or less.

QUESTIONS AND ANSWERS

1.

```
Q. In Microsoft Windows version 3.x, a temporary swap file is slower
   than a permanent swap file. Will letting Windows 95/98/Me set my
   virtual memory slow down my computer?

A. No. The temporary swap file in Windows 3.x has to switch between
   real mode and protected mode, slowing down the computer. As Windows
   95/98/Me runs only in protected mode, the swap file is as fast or
   faster than the permanent swap file in Windows 3.x.
```

2.

```
Q. I compressed my hard disk with DriveSpace. Can I still select my
   own virtual memory settings? In Windows 3.x I cannot use a
   permanent swap file on a compressed drive.
```

```
A. If your compressed drive is supported by a protected-mode driver,
   it is better to use the compressed drive for the paging file. If
   the compressed drive is not supported by a protected-mode driver
   then you must place the paging file on the host drive.
```

3.

```
Q. How can I determine whether my compressed drive is supported by a
   protected-mode driver?

A. At this time there are protected-mode drivers only for DoubleSpace
   and DriveSpace compressed drives. You can also check the IOS.INI
   file (if it exists) in the Windows directory to see which drives
   require real-mode support.
```

APPLIES TO

- Microsoft Windows 95

Keywords: kbdiskmemory kbinfo KB128327